PokerEquity

Brendan Fay and Lance Wong

December 2023

1 Introduction

We built a Poker Equity Calculator using the Monte Carlo method. The calculator is composed of two parts: the hand evaluator and the simulator. The hand evaluator uses a frequency calculator and pattern matching. There are more sophisticated implementations using lookup tables that are pretty exhaustive and very efficient, but our implementation is much more elegant. I have included segments from our code below; the implementation is accurate and the operations are not computationally expensive.

2 Implementation

We included Card, Hand, Rank, Suit, and HandRank algebraic data types. Card is composed of Suit and Rank, which are both ordinal and bounded. HandRank is also an ordinal data type which is ordered so that hands can be compared. Hand is a type alias for a list of Cards.

```haskell
data Suit = Spades | Hearts | Diamonds | Clubs
    deriving (Generic, Show, Eq)

data Rank =  Ace | King | Queen | Jack | Ten | Nine | Eight | Seven | Six | Five | Four | Three | Two
    deriving (Generic, Show, Ord, Eq, Enum, Bounded)

data HandRank = StraightFlush | FourOfAKind | FullHouse | Flush | Straight | ThreeOfAKind | TwoPair | Pair | HighCard
    deriving (Eq, Show, Ord, Enum, Bounded)
```

The Ranking function returns a hand's associated HandRank and the hand in increasing frequency-sorted order. This will be useful for comparing hand strengths later. To determine HandRank, the Classify helper function (within Ranking) does exhaustive pattern matching.

```haskell
share :: [Hand] -> Float
share (x:xs)
    | or $ map ((< ranking x) . ranking) xs = 0
    | otherwise = 1 / (fromIntegral $ (length . filter ((== ranking x) . ranking)) (x:xs))
share _ = 0.0


ranking :: Hand -> (HandRank, [Rank])
ranking = (,) <$> classify' <*> f
    where f = map snd . sort . map ((,) <$> negate . length <*> rank . head) . groupBy (==) . sort
```

The share function takes in a list of poker hands and evaluates to a floating point number between zero and one, inclusive. This number signifies the user's share of the pot. The value is 1.0 if the user wins and 0.0 if the user does not. In the case of a tie, the pot is divided.

The Shuffle function shuffles the deck with the Fisher-Yates algorithm, which is linear in the length of the deck. The Deal function removes the two user cards from the deck, shuffles it, then deals the community cards and the other player cards.

```haskell
shuffle :: StdGen -> Deck -> Deck
shuffle gen deck = fst $ foldl shuffleStep ([], gen) deck
    where
        shuffleStep (shuffled, g) cardIndex =
            let (index, newGen) = randomR (0, length shuffled) g
                (front, back) = splitAt index shuffled
            in (front ++ [cardIndex] ++ back, newGen)
```

```haskell
deal :: StdGen -> Hand -> [Card] -> Int -> Table
deal gen user community n = deal' shuffled
    where
        shuffled = community ++ shuffle gen deck
        deck = [Card s r | r <- [minBound..maxBound], s <- [Hearts,Diamonds,Clubs,Spades]] \\ complement
        complement = community ++ user
        deal' (a:b:c:d:e:fs) = [a,b,c,d,e] : user : (opponentCards n fs)
        deal' _ = []
        opponentCards 0 _ = []
        opponentCards m (x:y:zs) = [x, y] : (opponentCards (m - 1) zs)
        opponentCards _ _ = []
```

We evaluated our hands using a classification function, which pattern matches against the frequency of ranks in the best hand available to a player.

```haskell
classify' :: Hand -> HandRank
classify' hand =
    case groups of
        [1,1,1,1,1] -> case undefined of
                        _ | straight && flush   -> StraightFlush
                        _ | straight            -> Straight
                        _ | flush               -> Flush
                        _ | otherwise           -> HighCard
        [1,1,1,2]                               -> Pair
        [1,2,2]                                 -> TwoPair
        [1,1,3]                                 -> ThreeOfAKind
        [2,3]                                   -> FullHouse
        [1,4]                                   -> FourOfAKind
        _                                       -> HighCard
    where
        xs = (sort . map rank) hand
        (s:ss) = map suit hand
        straight = all (\(x, y) -> succ x == y) (pairwise xs)
        flush = all (== s) ss
        groups = (sort . map length . group) xs
        pairwise ls = zip ls (tail ls)
```

In order to run the Monte Carlo simulations, we pre-compute random number generators from the System.Random module with different seed values and run them each through one computation. The generators are used to shuffle the deck. We accumulate the total for each computation and return the average.
We chose the non-IO Monad random number generator because it made parallelization much easier. If we had used the IO Monad generator, the numbers that we generated would have been more random, in some sense. However, we considered the ability to parallelize our work more important.

```haskell
scoreRound :: Table -> Float
scoreRound (community:players) = share $ (map (bestHand . (++ community))) players
scoreRound _ = 0.0


playRound :: StdGen -> Hand -> [Card] -> Int -> Float
playRound gen user community players =
    scoreRound (deal gen user community players)
```

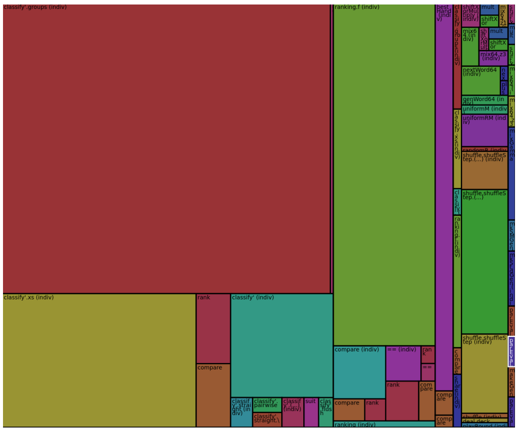
```haskell
monteCarlo :: Int -> Hand -> [Card] -> Int -> Float -> Float
monteCarlo n user community players pot =
    let results = (map (\g -> playRound g user community players) (makeGenerators n)) in
    pot * (sum results / (fromIntegral n))
```

The computation of the Monte Carlo simulations, and specifically the list in the results variable, are what we are attempting to parallelize. We also parallelized the computation of the generators, but that is not a huge asset to our execution time. In our benchmarking, we generated 10,000 generators and computed one experiment with each generator.

3 Parallel Computation

NOTE: All of the benchmarking that we did was of the individual functions, not the program as a whole.

Our sequential implementation takes about 1.7s to run a 10,000 experiment simulation. We can see a chart showing the time-share of each function and sub-function below (this was generated using profiteur):

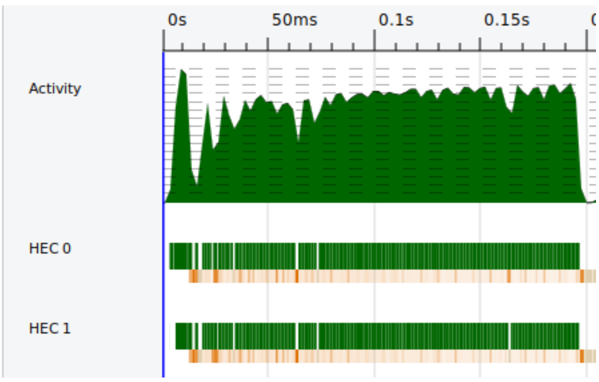


The labels are too small to read, but the boxes on the left side of the image represent the classify and ranking functions, while the boxes on the right represent a melange of other functions.

The most important information here is that the random number generation does not take a very long amount of time. We parallelized that aspect, but it did not contribute too much to our runtime improvements. Our major speedups came from parallelizing the computations themselves.

Naive Parallelization:

The parallelization step here breaks up the experiments naively, sparking a thread for each experiment. This gave a threefold speedup, profiling information is pictured below (in the interest of space, most of the HECs are going to be omitted):

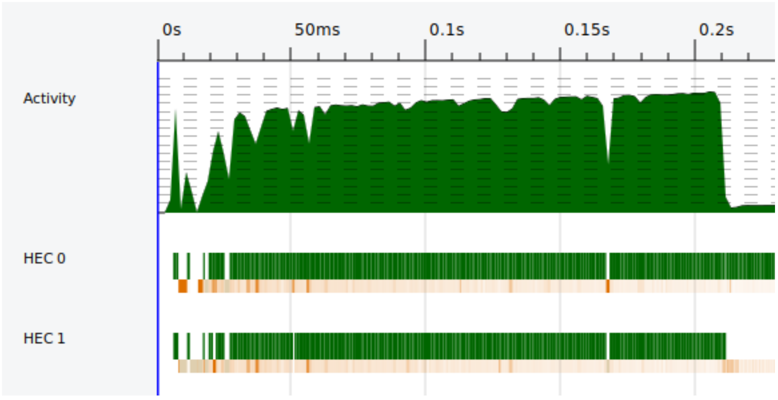


Threadscope shows that the activity in all of the cores is generally strong.

Chunked Parallelization:

The parallel step here breaks the experiments into chunks, and runs each of those chunks in parallel. This ran a little faster, but activity was a little lower. When we enforced the strict evaluation of the chunks, the activity increased to use more of the CPU. This caused a ten percent speedup over the naive version.

Threadscope shows similar activity to the previous method, with better activity overall. Originally, activity was worse with this method. When we forced strict evaluation to WHNF in the parallel step using rseq, the activity became better.

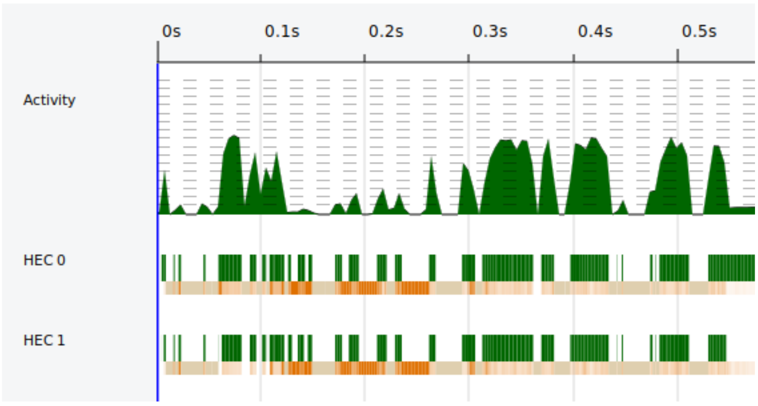


StdGen Splitting:

This parallelization method involved changing the way that we compute random numbers. Previously, we generated random numbers ahead of time. Our reasoning for this was that RNG would be less interesting to parallelize.

There exists a split function that takes one StdGen generator and returns two. This can be used to iterate random number generation. We decided to use this to divide the work. We first split a random number generator. We then spark a thread to evaluate a single experiment using rseq, and recurse on the rest of the experiments.

This produced a mild speedup, but not as much as either of the previous experiments. The profiling details are pictured below:

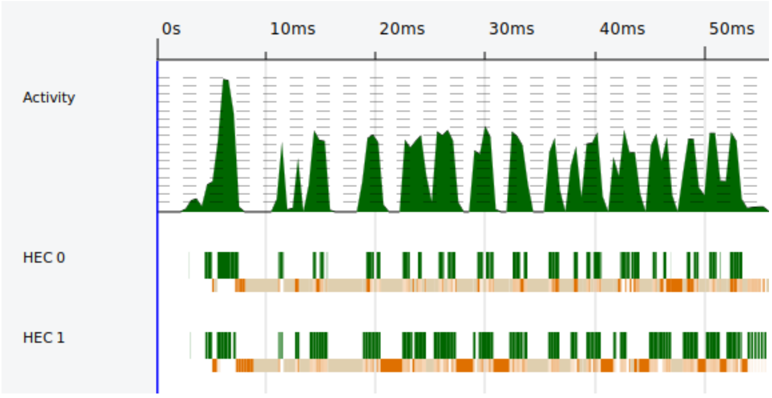


The activity is very choppy. This ran in about 1.09s, compared to 1.7s for the sequential implementation and .4s for the chunked implementation. It makes sense that the speedup would be slower, since a lot of the parallelism is devoted to computing inexpensive operations like making random number generators.

Chunking StdGen Splitting:

We decided to break our recursion into chunks of ten experiments. We would strictly evaluate a chunk in parallel, and spark a thread to recurse on it in parallel. This resulted in computations that took about .2 seconds on average. This is nearly a tenfold speedup from our sequential implementation.

The profiling is included below:



Once again, the profiling is extremely choppy. I suspect that there may be some compiler optimization occurring to cause this behavior. We have, at times, noticed fluctuations up to .2s with this implementation.

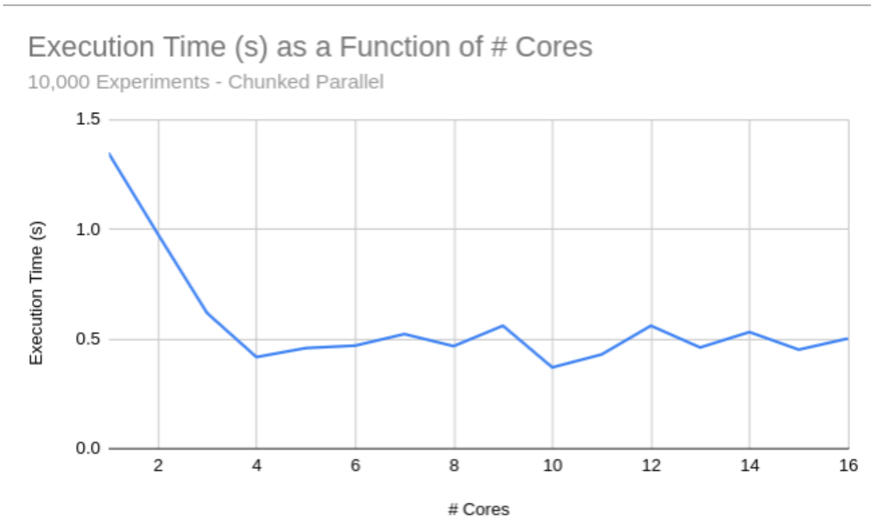
The activity with the recursive implementation is much less uniform than the parallel implementation. Also: the answers we get are different. This is because the non-IO Monad bound StdGen type is deterministic, so splitting vs. pre-computation is a very significant difference.

4 Conclusion

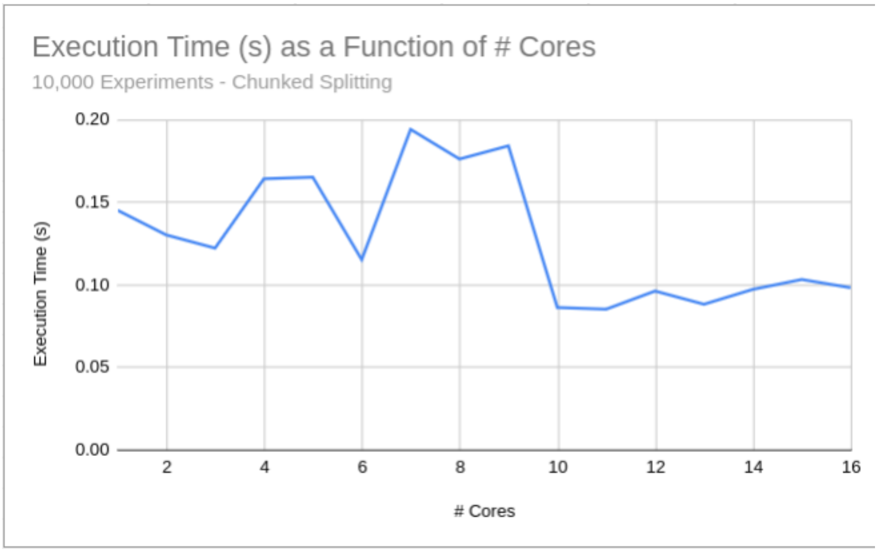
We tried four different methods of parallelization. Among the standard methods that were discussed in class, breaking the list into chunks and running each chunk in parallel seems like the best strategy. This gave us the best use of the CPU and the fastest run time.

We also tried an ad-hoc parallelism strategy by splitting random number generators as we go along. This was slower than naive parallelism when done experiment-by-experiment, but was faster than chunked parallelism when chunking. It is not obvious to us whether this has anything to do with the semantics of the split function, or whether it is a genuine result.

Generally, we saw an increase in the number of cores result in faster execution times. This speedup became somewhat negligible at around four cores. Pictured are the results of ten bench-markings at each level of granularity, and the results are born out in the chart.



The below chart shows the recursive splitting behavior when we chunked the experiments. These results are a little unusual and affirm some of our concerns about the legitimacy of this tenfold speedup. It seems like increasing the number of cores generally led to a slight speedup, but all of the speeds were close enough to each other that small differences look quite significant.



Pictured below are our overall execution times, measured with the -N flag.

Execution Speed

	10000 experiments
Sequential	1.934s
Naive Parallelization	.631s
Chunked Parallelization	.598s
Recursive Parallelization	1.53s
Recursive Chunked Parallelization	.135s

5 Code

```haskell
{-# LANGUAGE DeriveGeneric #-}
module Main where

-- for building and running instructions
-- https://github.com/bfay1/PokerEquity

import Data.List
import Data.Ord
import System.Random
import Control.Parallel.Strategies
import GHC.Generics
import System.Environment

stor :: String -> Rank
stor "Two" = Two
stor "Three" = Three
stor "Four" = Four
stor "Five" = Five
stor "Six" = Six
stor "Seven" = Seven
stor "Eight" = Eight
stor "Nine" = Nine
stor "Jack" = Jack
stor "Queen" = Queen
stor "King" = King
stor "Ace" = Ace
stor _ = Ace

stos :: String -> Suit
stos "Hearts" = Hearts
stos "Diamonds" = Diamonds
stos "Clubs" = Clubs
stos "Spades" = Spades
stos _ = Spades

stoh :: (String, String) -> Card
stoh (s, r) = Card (stos s) (stor r)

data Suit = Spades | Hearts | Diamonds | Clubs
    deriving (Generic, Show, Eq)

data Rank =  Ace | King | Queen | Jack | Ten | Nine | Eight | Seven | Six | Five | Four | Three | Two
    deriving (Generic, Show, Ord, Eq, Enum, Bounded)

data HandRank = StraightFlush | FourOfAKind | FullHouse | Flush | Straight | ThreeOfAKind | TwoPair | Pair | HighCar
    deriving (Eq, Show, Ord, Enum, Bounded)

ranks :: [Rank]
ranks = [Ace, King, Queen, Jack, Ten, Nine, Eight, Seven, Six, Five, Four, Three, Two]

suits :: [Suit]
suits = [Spades, Diamonds, Hearts, Clubs]


data Card = Card { suit :: Suit, rank :: Rank }


```

```haskell

instance Eq Card where
     x == y = rank x == rank y

instance Ord Card where
    x `compare` y = rank x `compare` rank y

instance Show Card where
    show (Card s r) = show r ++ " of " ++ show s


type Hand = [Card]
type Deck = [Card]
type Table = [[Card]]


share :: [Hand] -> Float
share (x:xs)
      | or $ map ((< ranking x) . ranking) xs = 0
      | otherwise = 1 / (fromIntegral $ (length . filter ((== ranking x) . ranking)) (x:xs))
share _ = 0.0


ranking :: Hand -> (HandRank, [Rank])
ranking = (,) <$> classify' <*> f
    where f = map snd . sort . map ((,) <$> negate . length <*> rank . head) . groupBy (==) . sort


classify' :: Hand -> HandRank
classify' hand =
    case groups of
        [1,1,1,1,1] -> case undefined of
                            _ | straight && flush   -> StraightFlush
                            _ | straight            -> Straight
                            _ | flush               -> Flush
                            _ | otherwise           -> HighCard
        [1,1,1,2]                                   -> Pair
        [1,2,2]                                     -> TwoPair
        [1,1,3]                                     -> ThreeOfAKind
        [2,3]                                       -> FullHouse
        [1,4]                                       -> FourOfAKind
        _                                           -> HighCard
    where
        xs = (sort . map rank) hand
        (s:ss) = map suit hand
        straight = all (\(x, y) -> succ x == y) (pairwise xs)
        flush = all (== s) ss
        groups = (sort . map length . group) xs
        pairwise ls = zip ls (tail ls)

```

```haskell
shuffle :: StdGen -> Deck -> Deck
shuffle gen deck = fst $ foldl shuffleStep ([], gen) deck
    where
        shuffleStep (shuffled, g) cardIndex =
            let (index, newGen) = randomR (0, length shuffled) g
                (front, back) = splitAt index shuffled
            in (front ++ [cardIndex] ++ back, newGen)


deal :: StdGen -> Hand -> [Card] -> Int -> Table
deal gen user community n = deal' shuffled
    where
        shuffled = community ++ shuffle gen deck
        deck = [Card s r | r <- [minBound..maxBound], s <- [Hearts,Diamonds,Clubs,Spades]] \\ complement
        complement = community ++ user
        deal' (a:b:c:d:e:fs) = [a,b,c,d,e] : user : (opponentCards n fs)
        deal' _ = []
        opponentCards 0 _ = []
        opponentCards m (x:y:zs) = [x, y] : (opponentCards (m - 1) zs)
        opponentCards _ _ = []


userHand :: Hand
userHand = [Card Diamonds Ace, Card Hearts Ace]

bestHand :: [Card] -> Hand
bestHand cards = minimumBy (comparing ranking) $ filter ((==5) . length) (subsequences cards)

scoreRound :: Table -> Float
scoreRound (community:players) = share $ (map (bestHand . (++ community))) players
scoreRound _ = 0.0


playRound :: StdGen -> Hand -> [Card] -> Int -> Float
playRound gen user community players =
    scoreRound (deal gen user community players)


makeGenerators :: Int -> [StdGen]
makeGenerators n = runEval $ parList rseq (map mkStdGen [50..(50 + n - 1)])


monteCarlo :: Int -> Hand -> [Card] -> Int -> Float -> Float
monteCarlo n user community players pot =
    let results = (map (\g -> playRound g user community players) (makeGenerators n)) in
    pot * (sum results / (fromIntegral n))


parallelMonteCarlo :: Int -> Hand -> [Card] -> Int -> Float -> Float
parallelMonteCarlo n user community players pot =
    let results = runEval $ parList rseq (map (\g -> playRound g user community players) (makeGenerators n)) in
    pot * (sum results / (fromIntegral n))
```

```haskell
recursiveMonteCarlo :: StdGen -> Int -> Hand -> [Card] -> Int -> Float -> [Float]
recursiveMonteCarlo _ 0 _ _ _ _ = []
recursiveMonteCarlo gen n user community players pot =
    let (gen1, gen2) = split gen
    in (runEval $ rseq $ playRound gen1 user community players) : (runEval $ rseq $ recursiveMonteCarlo gen2 (n - 1) use

recursiveChunkMonteCarlo :: StdGen -> Int -> Hand -> [Card] -> Int -> Float -> [Float]
recursiveChunkMonteCarlo _ 0 _ _ _ _ = []
recursiveChunkMonteCarlo gen n user community players pot =
    let (gen1, gen2) = split gen
    in ( {- runEval $ parList rseq $ -} replicate 10 (runEval $ rseq $ playRound gen2 user community players)) ++ (runEv

-- Create a fixed number of RNGs
makeFixedRNGs :: Int -> [StdGen]
makeFixedRNGs m = map mkStdGen [50..(49 + m)]

-- Function to divide experiments into chunks
divideIntoChunks :: Int -> [a] -> [[a]]
divideIntoChunks _ [] = []
divideIntoChunks n xs = take n xs : divideIntoChunks n (drop n xs)

-- Parallel Monte Carlo function using a fixed number of RNGs
parallelMonteCarloFixedRNGs :: Int -> Int -> Hand -> [Card] -> Int -> Float -> Float
parallelMonteCarloFixedRNGs m n user community players pot =
    let rngs = cycle (makeFixedRNGs m)  -- Creates a repeating list of RNGs
        results = runEval $ parList rseq [playRound gen user community players | gen <- take n rngs]
    in pot * ((sum $ runEval $ parList rseq results) / fromIntegral n)

-- Function to run a chunk of experiments
runChunk :: (StdGen, [(Hand, [Card], Int)]) -> [Float]
runChunk (gen, experiments) =
    map (\(user, community, players) -> runEval $ rseq (playRound gen user community players)) experiments

initialDeck :: [Card]
initialDeck = [Card s r | r <- [minBound..maxBound], s <- [Hearts,Diamonds,Clubs,Spades]]


sequential :: Int -> Hand -> Int ->  Float
sequential e hand players = monteCarlo e hand [] players 100

naive :: Int -> Hand -> Int -> Float
naive e hand players = parallelMonteCarlo e hand [] players 100

chunk :: Int -> Hand -> Int -> Float
chunk e hand players = parallelMonteCarloFixedRNGs 30 e hand [] players 100

recursive :: Int -> Hand -> Int -> Float
recursive e hand players =
    let gen = mkStdGen 42
        total = sum $ runEval $ parList rseq (recursiveMonteCarlo gen e hand [] players 100) in
    (total / fromIntegral e)
```

```haskell
recursiveChunk :: Int -> Hand -> Int -> Float
recursiveChunk e hand players =
    let gen = mkStdGen 42
        total = sum $ runEval $ parList rseq (recursiveChunkMonteCarlo gen e hand [] players 100) in
    (total / fromIntegral e)


findMethod :: String -> Int -> Hand -> Int -> Float
findMethod "sequential" e hand players = sequential e hand players
findMethod "naive" e hand players = naive e hand players
findMethod "chunk" e hand players = chunk e hand players
findMethod "recursive" e hand players = recursive e hand players
findMethod "recursiveChunk" e hand players = recursiveChunk e hand players
findMethod _ _ _ _ = 0.0

main :: IO ()
main = do
    args <- getArgs
    if length args /= 7
        then putStrLn $ "Usage : ./Main <suit> <rank> <suit> <rank> <numplayers> <numexperiments> <method>"
    else do
        let [s1, r1, s2, r2, players, experiments, method] = args
        putStrLn $ show $ findMethod method (read experiments) [(stoh (s1, r1)), (stoh (s2, r2))] (read players)
```